Modular Exponentiation

Modular exponentiation is the problem of finding an efficient way of computing $a^b \bmod n$. Modular exponentiation is applicable in many security measures, such as RSA. The naive approach to computing $a^b \bmod n$ would be to calculate $a \bmod n$ and then multiplying the result by a an additional $b-1$ times. This method requires using $O(b)$ multiplications to get the intended result.

One key observation to make is that we can create shortcuts by squaring results instead of multiplying by a one at a time. For example, if we want to calculate $a^10 \bmod n$ and we've already calculated $a^5 \bmod n$, instead of multiplying the result by a 5 more times, we can square a^5 to get a^10, getting to our intended result in just one multiplication instead of five. Squaring doubles the exponent while multiplying increases the exponent by 1. Using a combination of squaring and multiplying will result in modular exponentiation using $O(\log b)$ multiplications to get the intended result.

To figure out what order of squaring/multiplying we want to execute, it helps to take a look at the binary representation of b. Given a binary representation of b, $(b_k, b_{k-1}, ..., b_1, b_0)$, we iterate through the digits of b from most significant to least significant. Every time we see a 1 digit, that means we must multiply. Every time we see a 0 digit, we don't multiply. Then, to shift to the next binary digit, we square the result. For example, to solve $a^b \bmod n$ where $b = 10$, we break down b into its binary representation 1010. Then we just iterate through the binary representation:

$$\text{Start with } 1 \bmod n \tag{1}$$
$$b_3 = 1, (1 * a) \bmod n = a \bmod n \tag{2}$$
$$\text{Shift to } b_2, (a)^2 \bmod n = a^2 \bmod n \tag{3}$$
$$b_2 = 0, a^2 \bmod n \tag{4}$$
$$\text{Shift to } b_1, (a^2)^2 \bmod n = a^4 \bmod n \tag{5}$$
$$b_1 = 1, (a^4 * a) \bmod n = a^5 \bmod n \tag{6}$$
$$\text{Shift to } b_0, (a^5)^2 \bmod n = a^{10} \bmod n \tag{7}$$
$$b_0 = 0, a^{10} \bmod n \tag{8}$$

Parallel Addition

Adding two n-bit numbers the naive way involves adding the least significant digits together, carrying over a 1 if necessary, then repeating for all the other digits iteratively until all n bits have been added together. In this method, we cannot compute the ith bit of the sum before computing all the bits after it first since the ith bit depends on whether or not there was a carry involved.

With a **carry lookahead adder**, we can add two n-bit numbers in $O(\lg n)$ time by taking advantage of some clever anticipation of carries and parallelization. For the purposes of this recitation though, we won't go over the inner details of how it does this, though you can look online if you're interested in more details.

Instead, we'll look at the **carry-save adder**, that allows us to reduce the problem of adding three n-bit numbers to the problem of adding two n-bit numbers efficiently. Say the problem is to add three n-bit numbers, i.e. $x + y + z$. We will reduce this to the problem of adding two n-bit numbers, $u + v$. u is going to represent the sum of x, y, z ignoring all carries. v is going to represent just the carries of x, y, z. For example:

```
  0 0 0 1 1 1 0 1 = x
  1 1 0 0 0 1 0 1 = y
  1 0 0 1 0 1 1 0 = z
  -------------------
  0 1 0 0 1 1 1 0 = u
1 0 0 1 0 1 0 1   = v
```

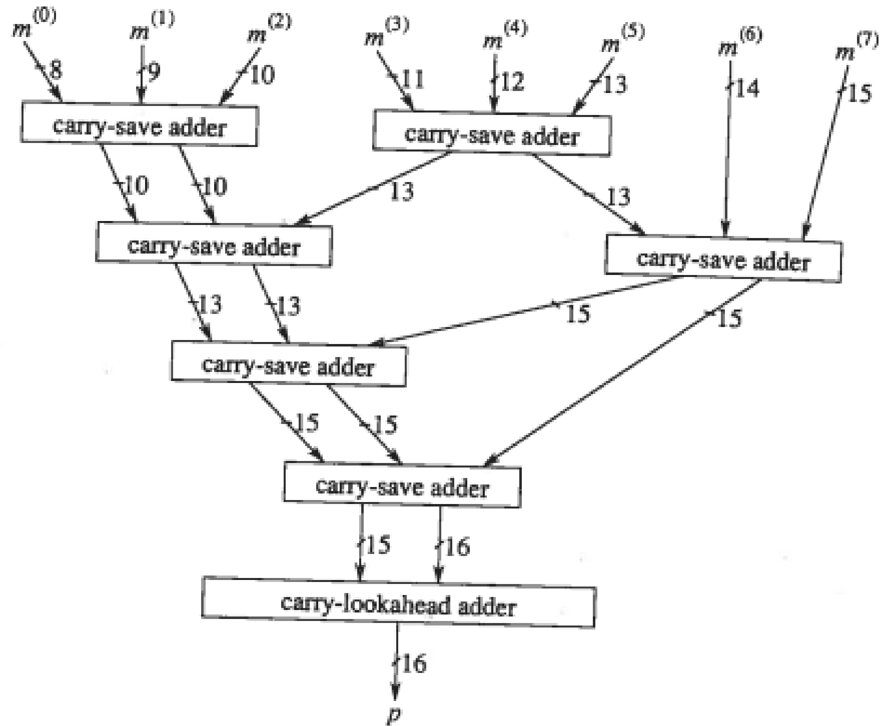
Note that u and v can be computed completely independently from each other. Before we were concerned about whether or not the sum would be impacted by carries or not, but now that we've completely separated the carries from the actual addition, we can compute u and v separately. Not only that, each digit of u and v can be computed separately.

- u_i = parity(x_i, y_i, z_i) (= $x_i + y_i + z_i$ mod 2)
- v_{i+1} = majority(x_i, y_i, z_i) (= 0 if there are more 0s than 1s, =1 if there are more 1s than 0s)

With this observation, if we have n separate processes that can compute parity and majority, we can compute u and v in $O(1)$ time since u and v both have $O(n)$ bits. Once we've reduced the three sum problem to the two sum problem, we can use our carry lookahead adder to finish the job. Note that even though we can make $O(1)$ time reductions to the two sum problem, our running time is still $O(\lg n)$ because of the bottleneck of the carry lookahead adder at the very end. Asymptotically, this isn't better than using two carry lookahead adders to calculate $x + y$ and then $(x + y) + z$, but practically it is faster since we're using fewer carry lookahead additions.

The speed up is even more apparent if we are trying to add n n-bit numbers together. Like before, we will try to make $O(1)$ time reductions until we have reduced the problem into adding two n-bit numbers together, from where we can use a carry lookahead adder to finish the job.

The idea is to use a **Wallace tree** to keep applying the same 3-to-2 sum reductions until the problem has been reduced to a sum of two numbers. At every step, if we have k sum problem, we can use approximately $k/3$ carry-save adders to reduce the problem to an approximately $2k/3$ sum problem. Below is an example of what the tree may look like.



Since each carry-save adder removes one sum from the problem, we will need $O(n)$ carry-save adders to reduce the problem down to a two sum problem. Then we apply a single $O(\lg n)$ time carry lookahead adder to finish the sum. Since all the carry-save adders take $O(1)$ time assuming n processors, the total runtime of summing n n-bit numbers is $O(n)$.